

FIG. 1

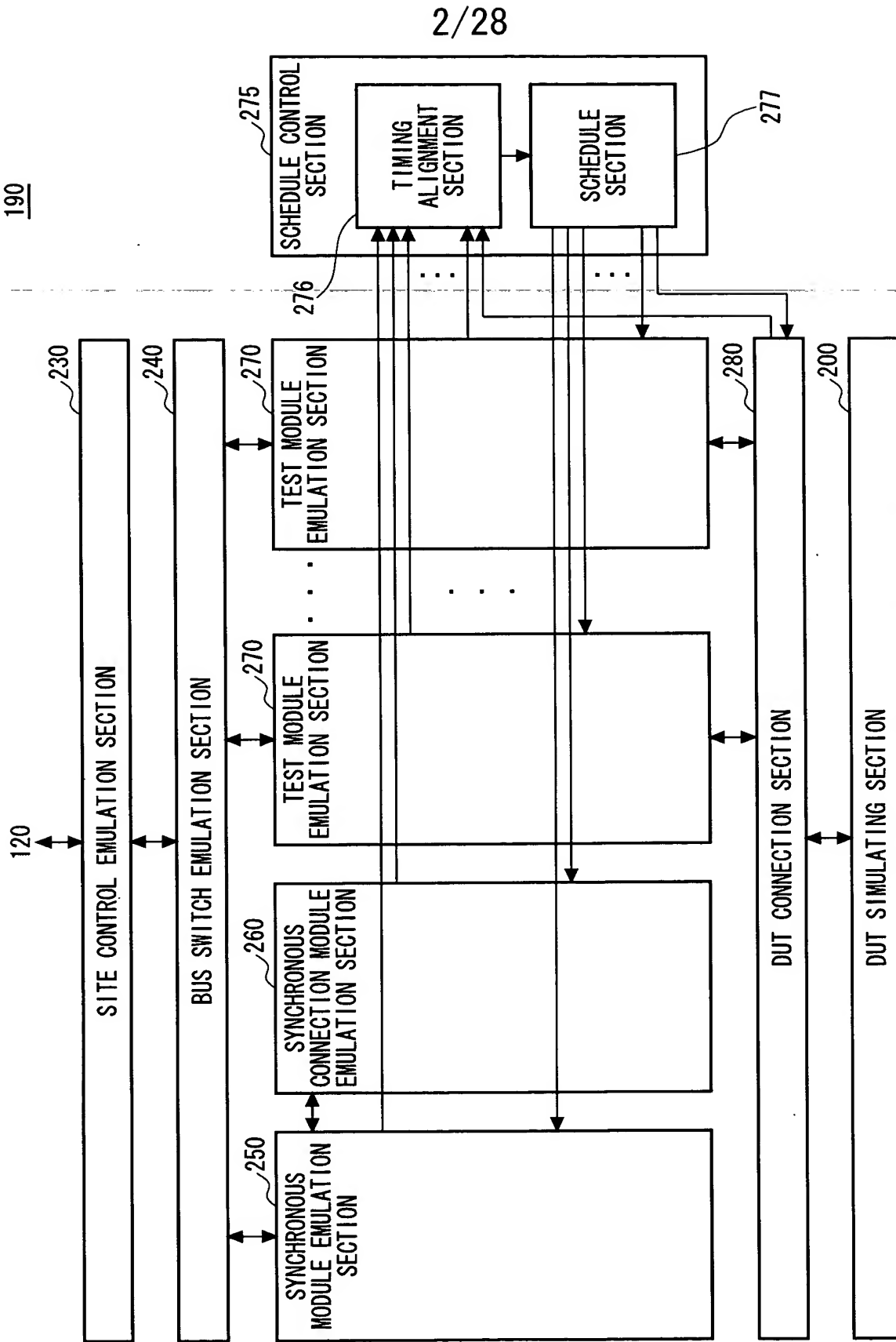


FIG. 2

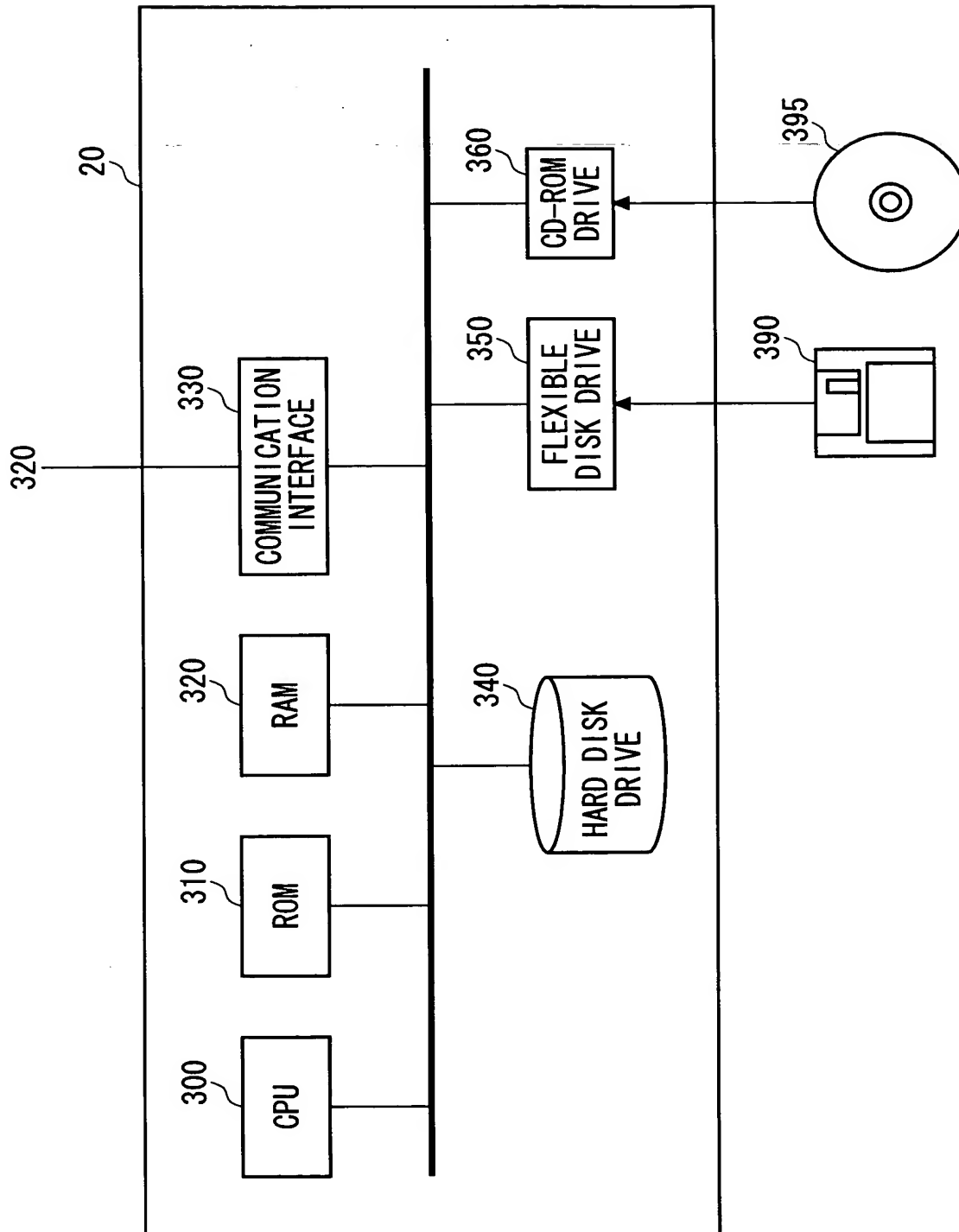


FIG. 3

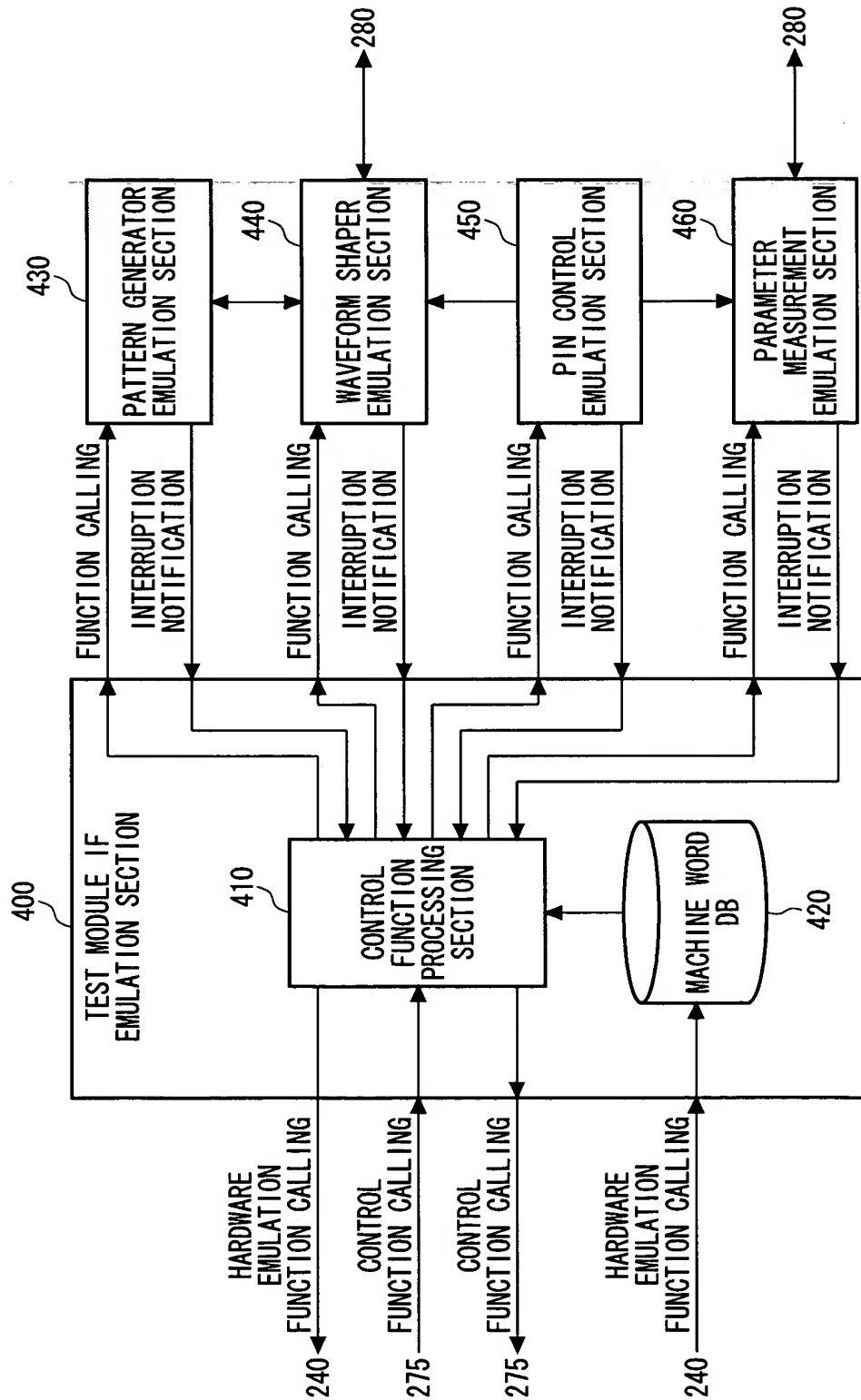
270

FIG. 4

500

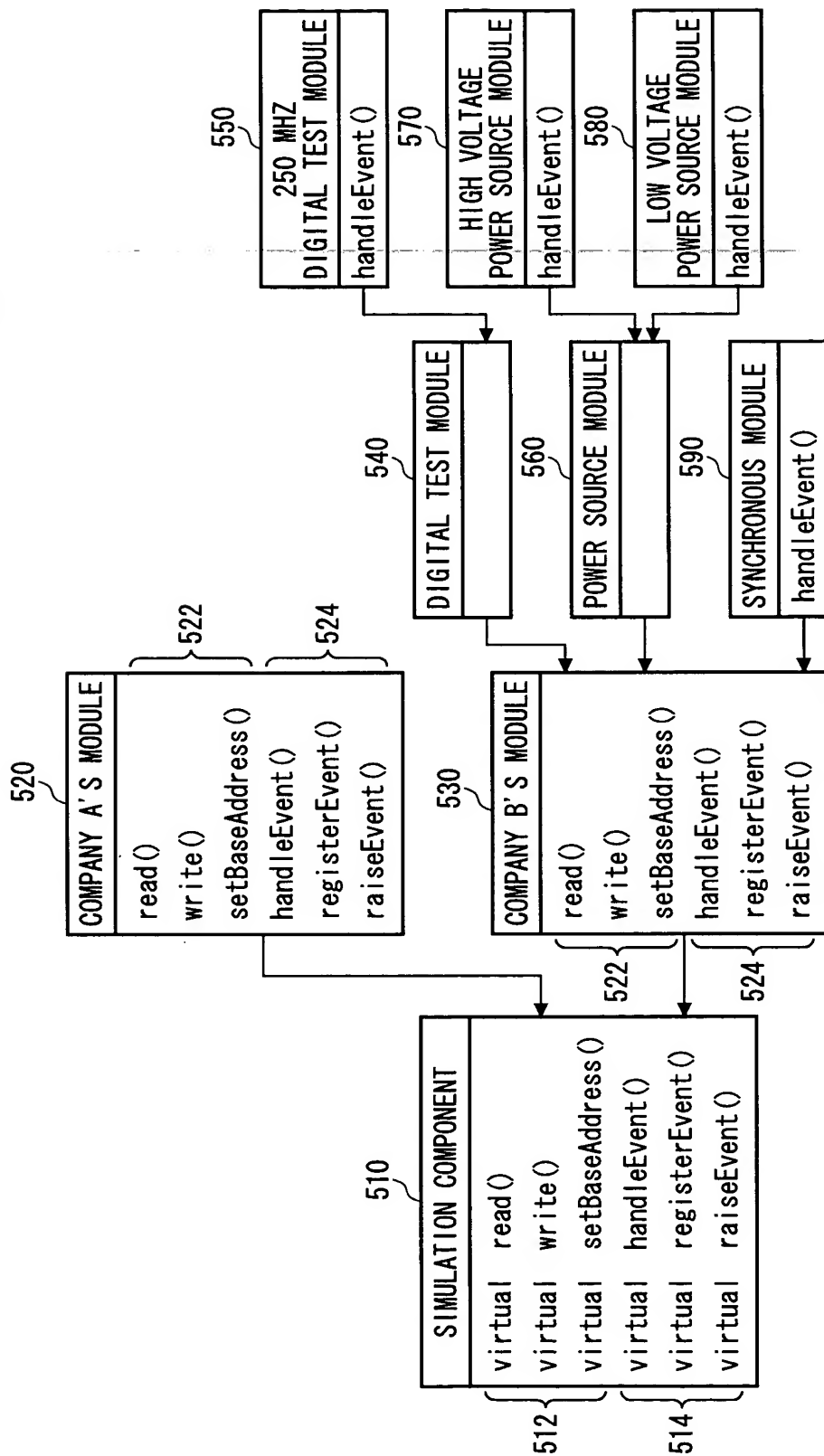


FIG. 5

6/28

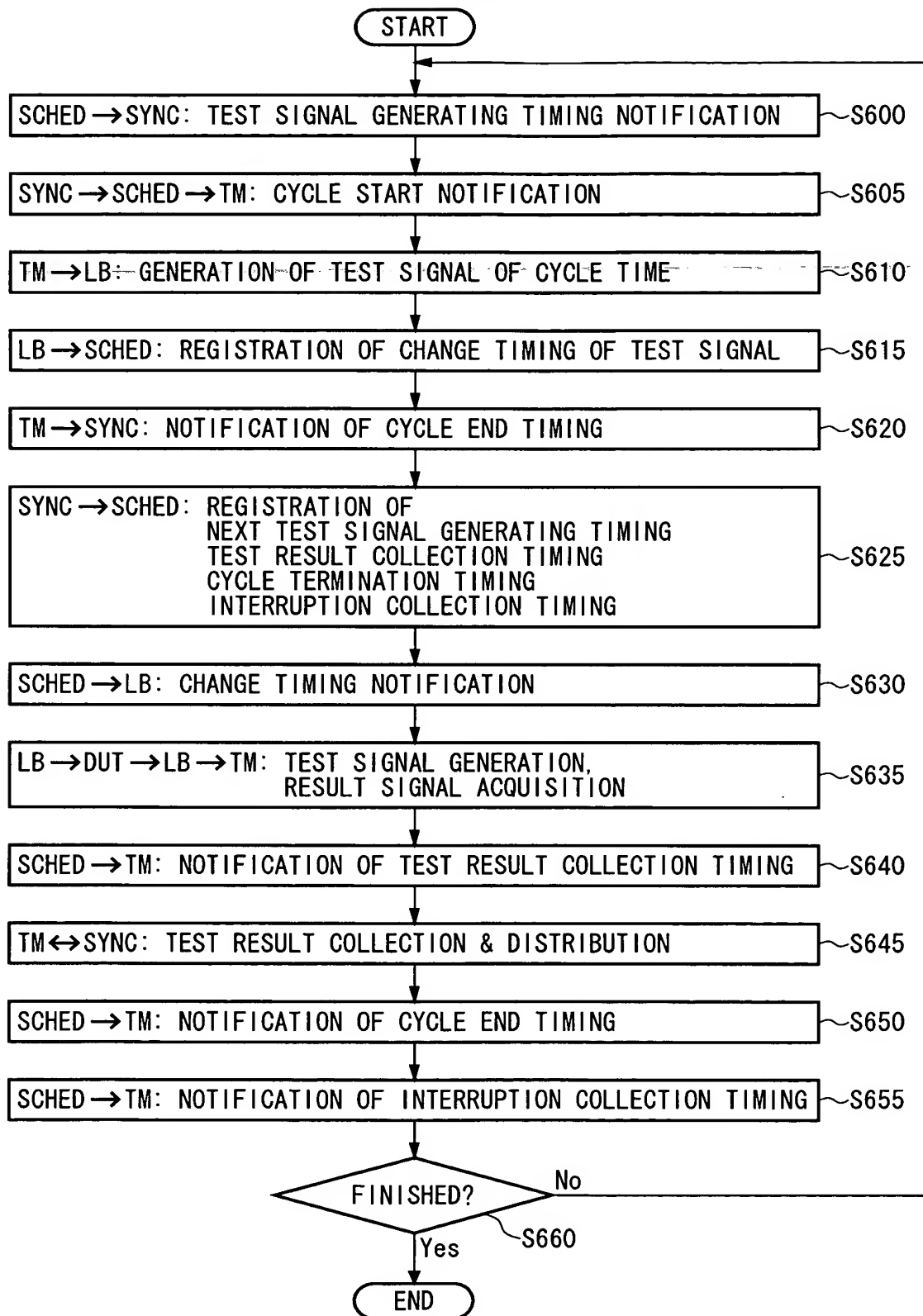


FIG. 6

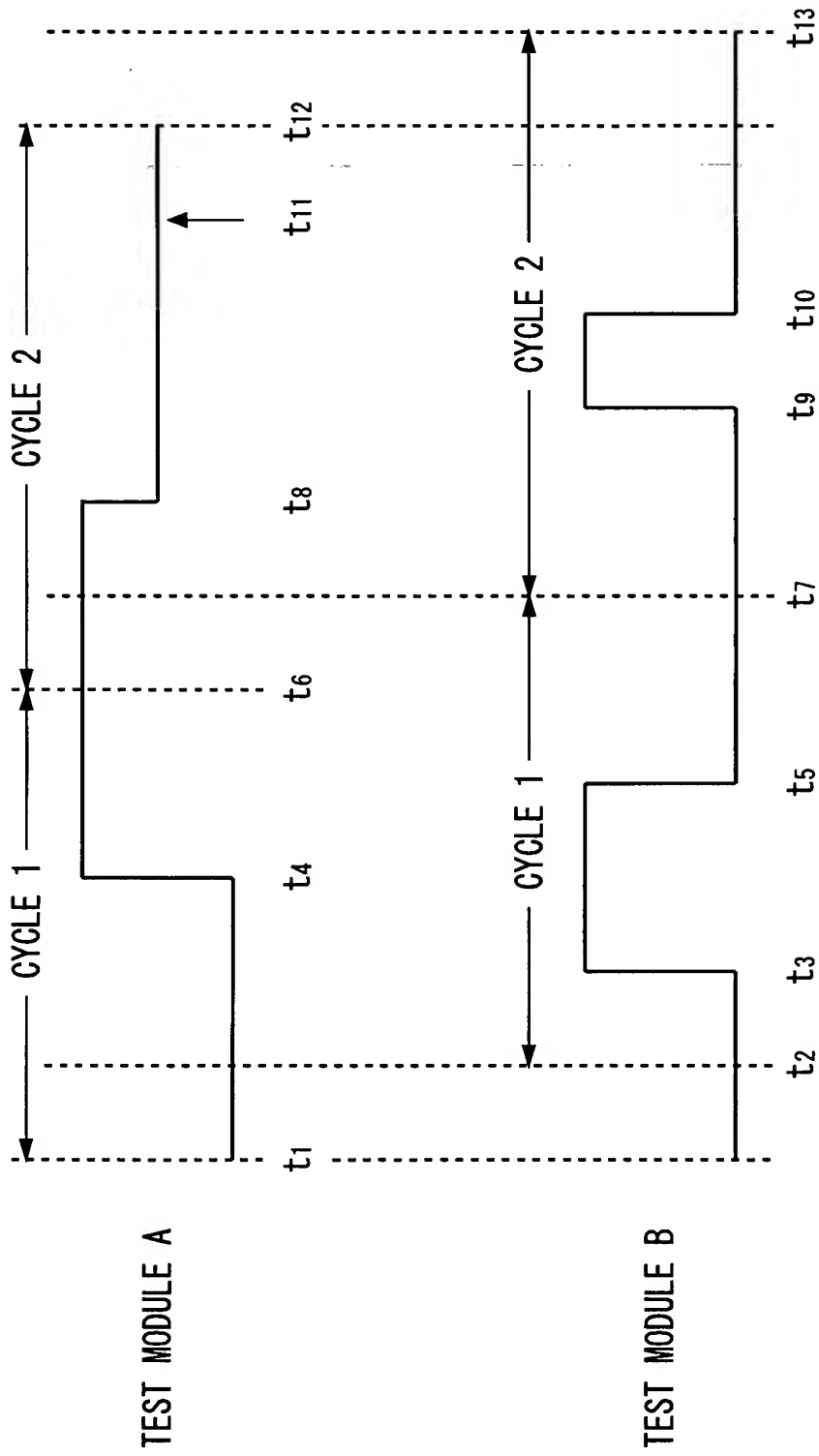


FIG. 7

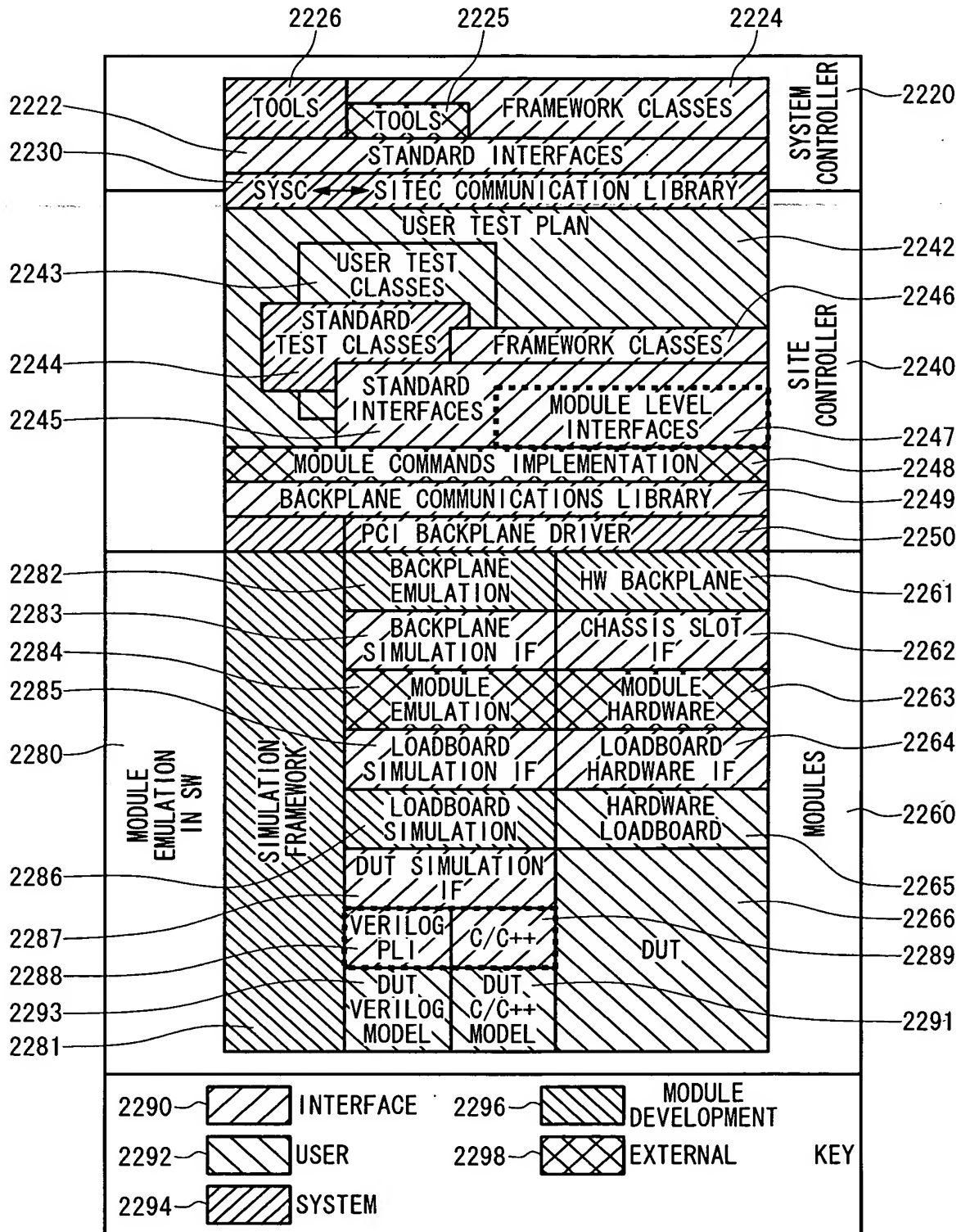
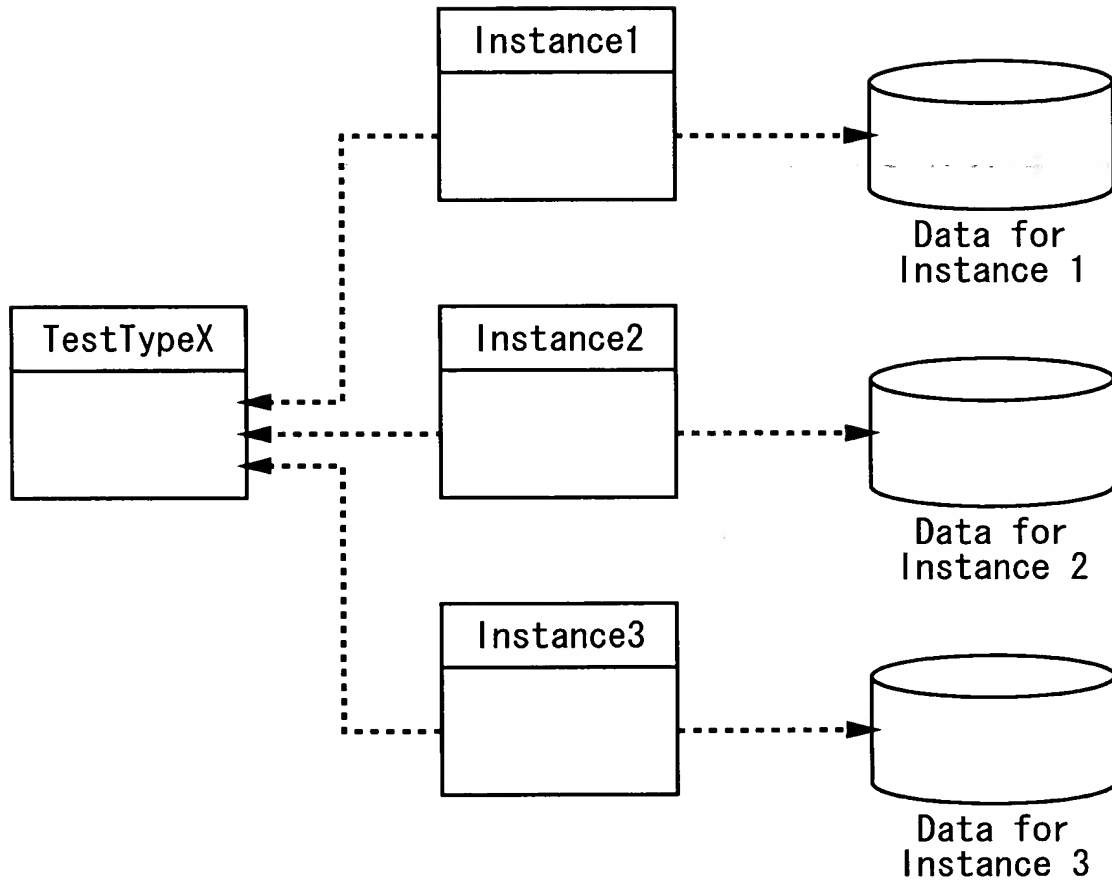


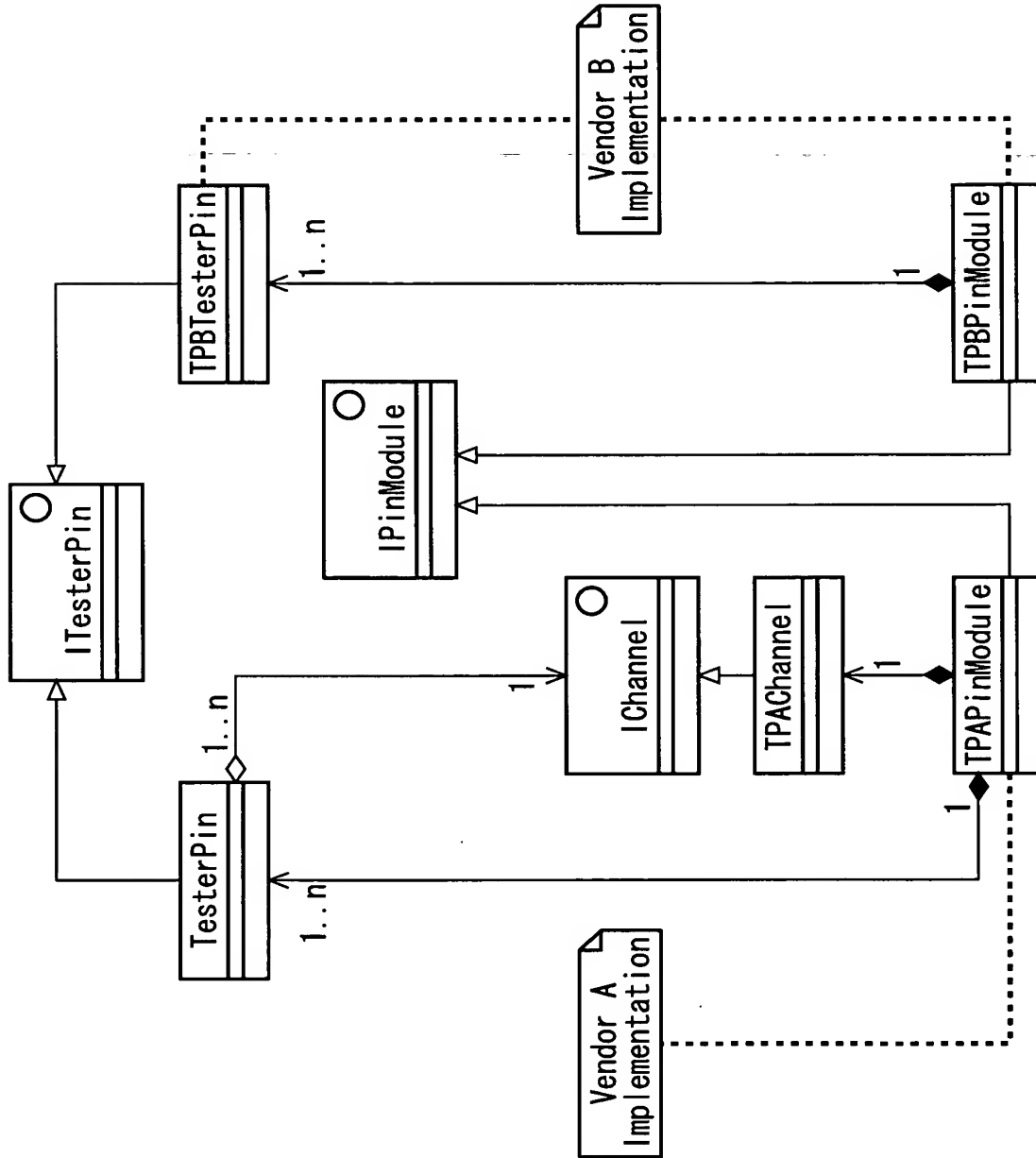
FIG. 8



9/28



*FIG. 9*



**FIG. 10**

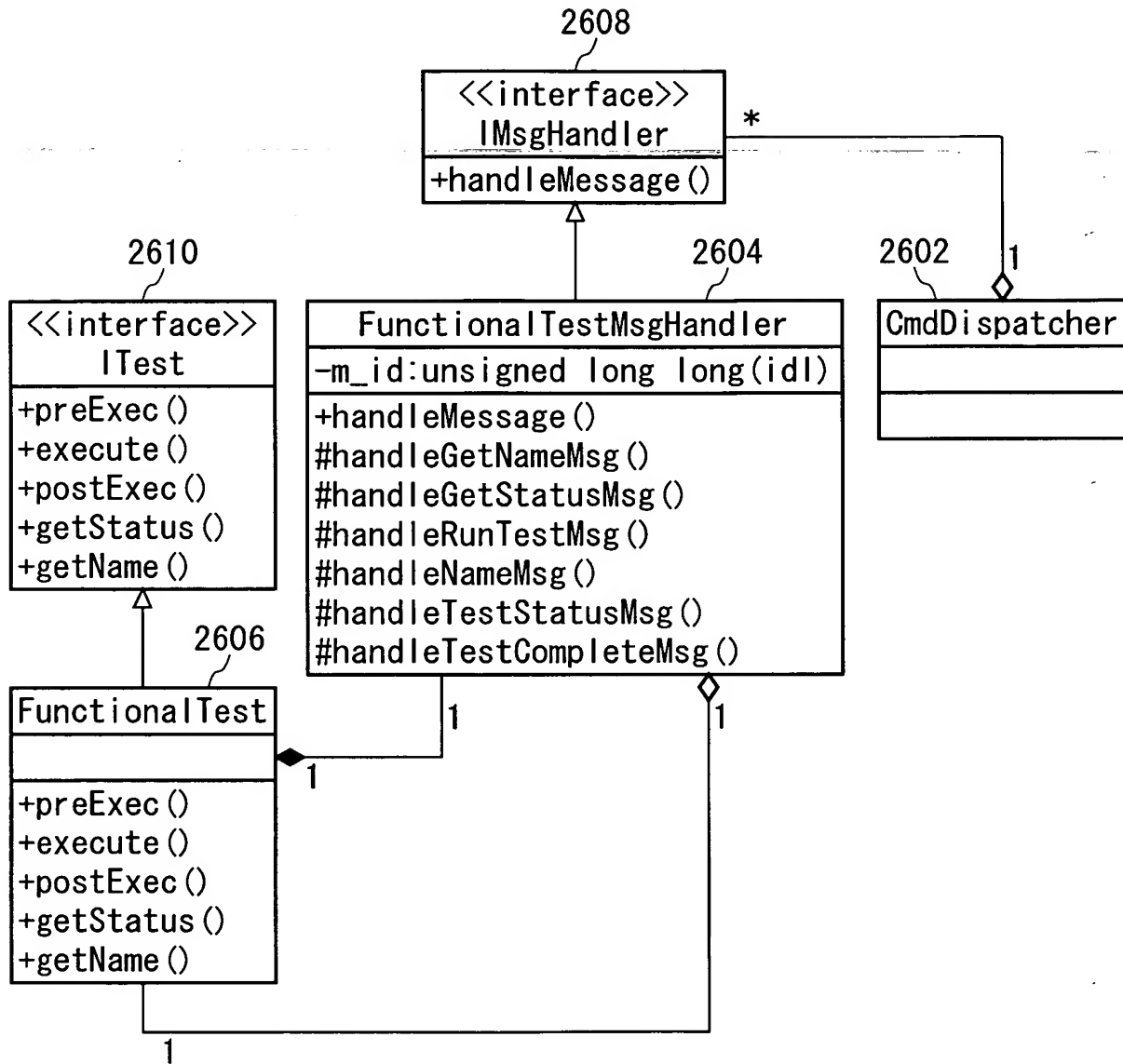


FIG. 11

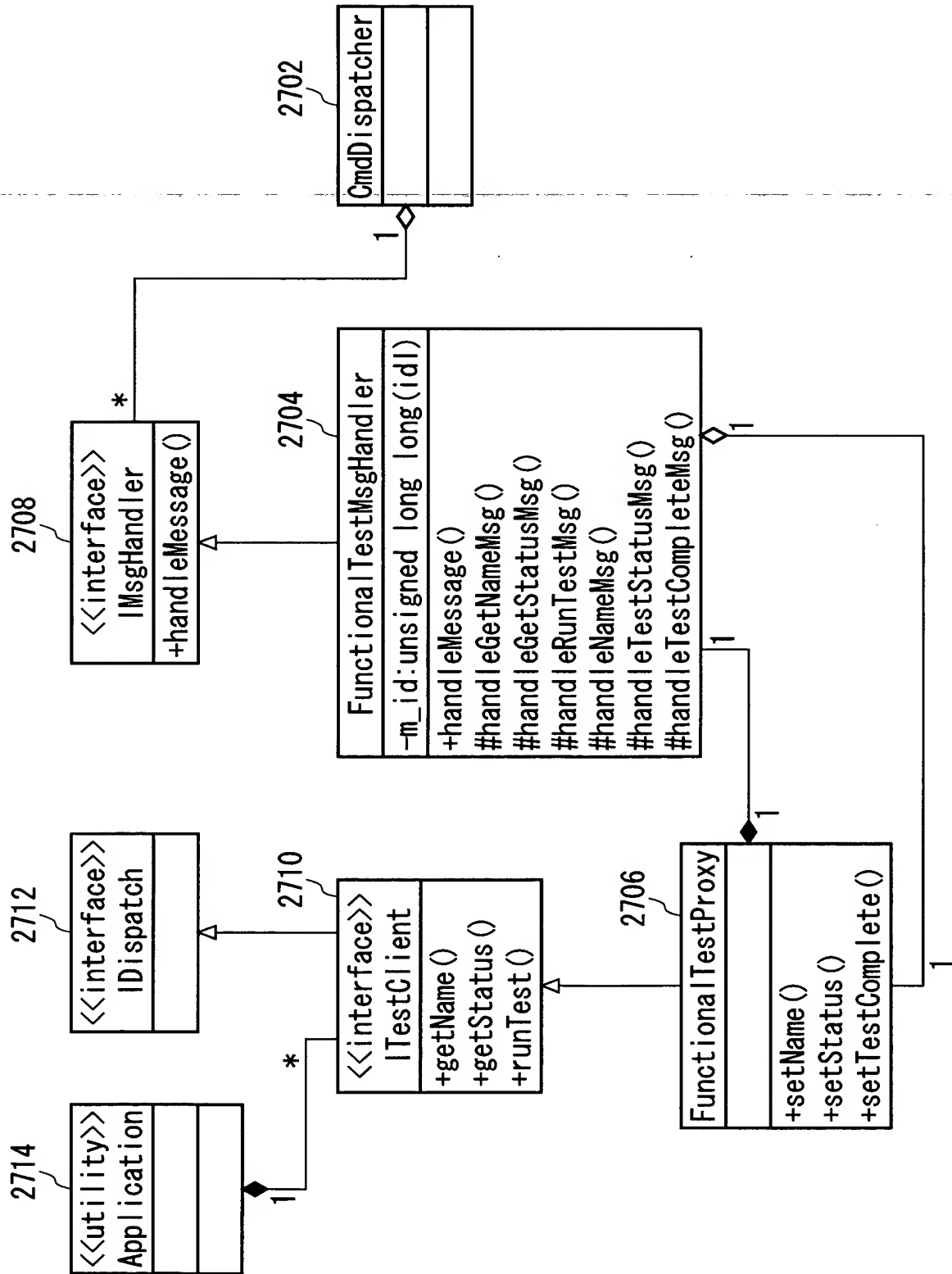


FIG. 12

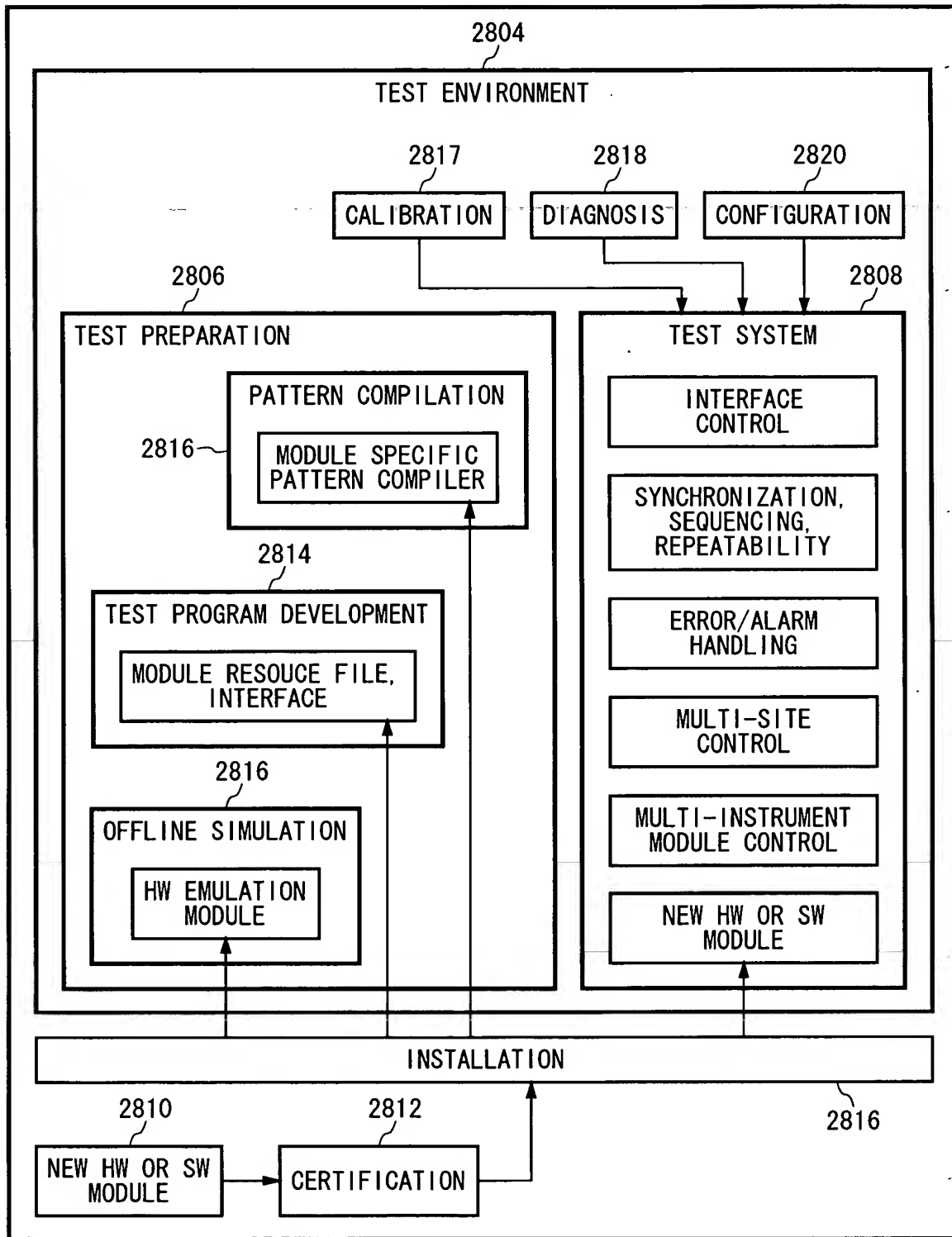


FIG. 13

```

Global
{
    InitVoltage 0.5 V;      # Required. Initial voltage on all
                           # wires.
    RecoveryRate 2.5 V/nS;  # Optional. For drive conflict in
                           # analog signals.
}
5010

# Module Emulator.
EMUModule "module1"      # Module DLL.
{
    Waveform              # Optional. Resource declaration.
    {
        Step 1-32, 35;    # Step type waveforms on channels 1 thru 32
                           # channel 35.
        Analog 33, 34;    # Analog waveform on channels 33 and 34.
    }
    Port 1                # Declares the GBUS Port for this module.
    {
        SerialNumber 1;   # Required. Should match setting in
                           # Module Configuration File.
        ProductRevision 1; # Required. Should match setting in
                           # Module Configuration File.
        Params            # To be passed to DLL.
        {
            test "param1";
            key  "abc";
        }
    }
    Port 8
    {
        LogicalPort 3;    # Optional. Designate Logical Port to use
                           # in offline configuration file.
                           # Default is the GBUS port.

        SerialNumber 2;
        ProductRevision 1;
        Params          # To be passed to DLL.
        {
            test "param1";
            key  "abc";
        }
    }
}
5020

```

FIG. 14

5000

```
# Module Emulator
EMUModule "module2"
{
    Waveform
    {
        Step 1-32;
    }
    Port 2
    {
        SerialNumber 1;
        ProductRevision 1;
    }
}
```

5020

```
# Module Emulator
EMUModule "dps"
{
    Waveform
    {
        Slew 1-32 @ 2.0 V / nS; # The slew rate is required
                                # for all slewing waveforms.
    }
    Port 4
    {
        SerialNumber 1;
        ProductRevision 1;
    }
}
```

5010

*FIG. 15*

5100

```
Global
{
  RegSelect "PatTraceSel"; # Pattern Tracing - Name of OASIS
                           # Simulated Register Selection File.
}

DUTModel "DUT1Sim"
{
  Waveform
  {
    Step 1-32;
  }
  DUT 1
  {
    Params
    {
      param1Name "param1Value";
      param2Name "param2Value";
    }
    PinConnections
    {
      L3.1      1      1.0 nS;
      L3.2      8      1.0 nS;
      L3.3      2      1.0 nS;
      L3.4      7      1.0 nS;
      L3.5      3      1.0 nS;
      L3.6      6      1.0 nS;
      L3.7      4      1.0 nS;
      L3.8      5      1.0 nS;
      L3.9      9      1.0 nS;
      L3.10     16     1.0 nS;
      L3.11     10     1.0 nS;
      L3.12     15     1.0 nS;
      L3.13     11     1.0 nS;
      L3.14     14     1.0 nS;
      L3.15     12     1.0 nS;
      L3.16     13     1.0 nS;
    }
  }
}
```

5110

5120

*FIG. 16*



```

DUTModel "DUT2Sim"
{
  Waveform
  {
    Slew 1-16 @ 2.0 V/nS;
  }
  DUT 2
  {
    Params
    {
      param1Name "param1Value";
      param2Name "param2Value";
    }
    PinConnections
    {
      L2.1 1; # no delay specified means a delay of 0
      L2.2 2;
      L2.3 3;
      L2.4 4;
      L2.12 5;
      L2.13 6;
      L2.14 7;
      L2.15 8;
    }
  }
}

```

*FIG. 17*

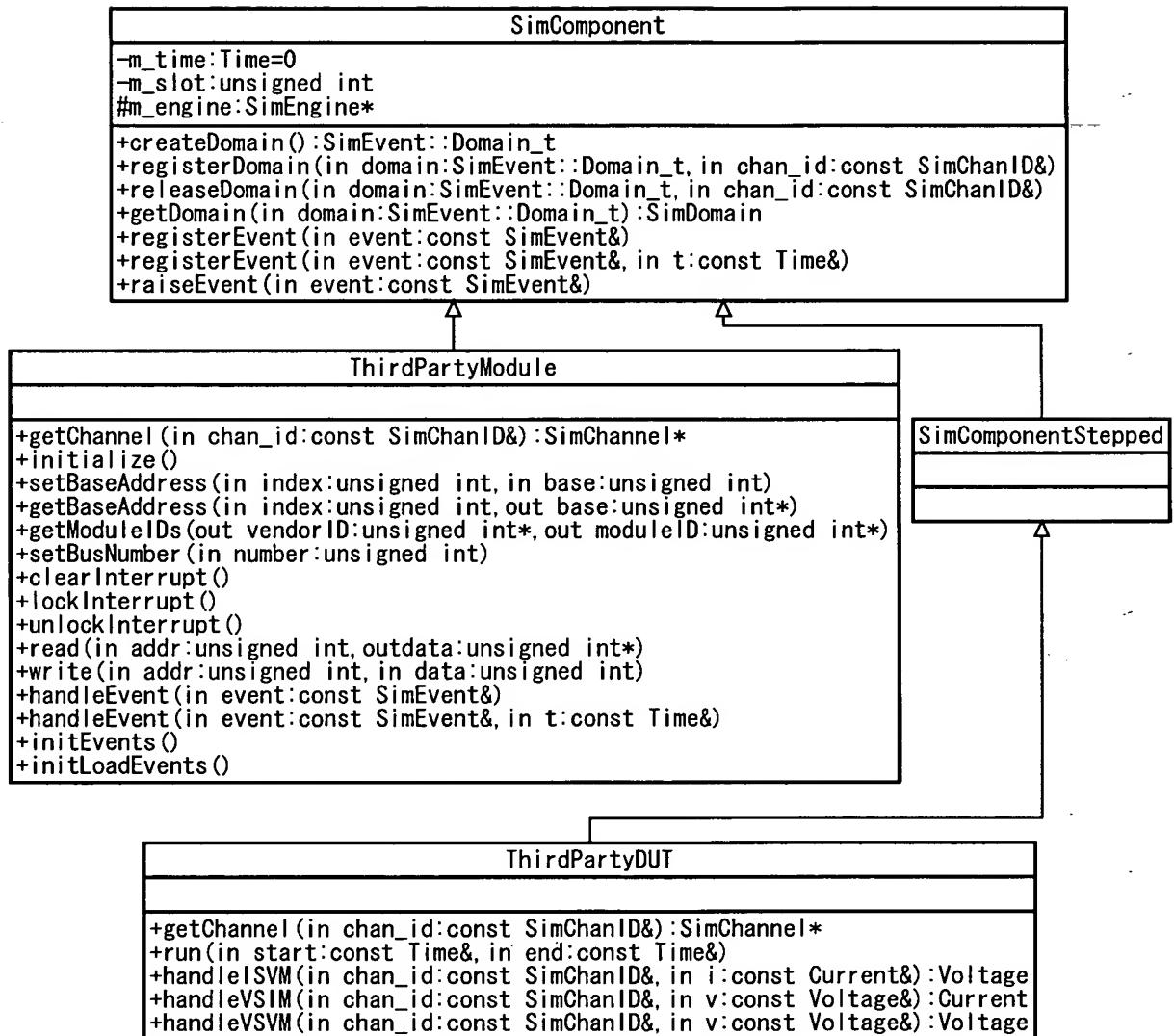


FIG. 18

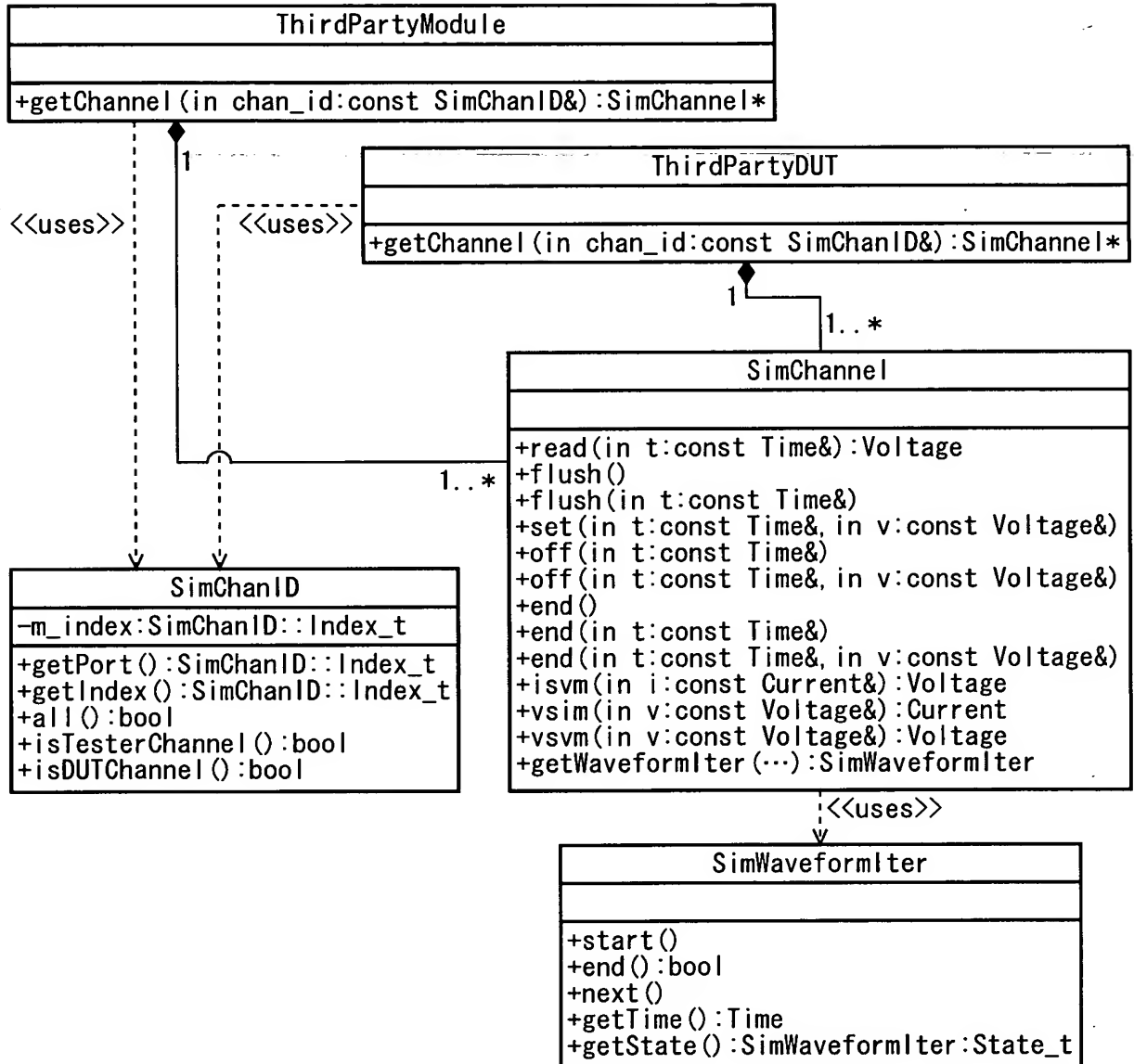


FIG. 19

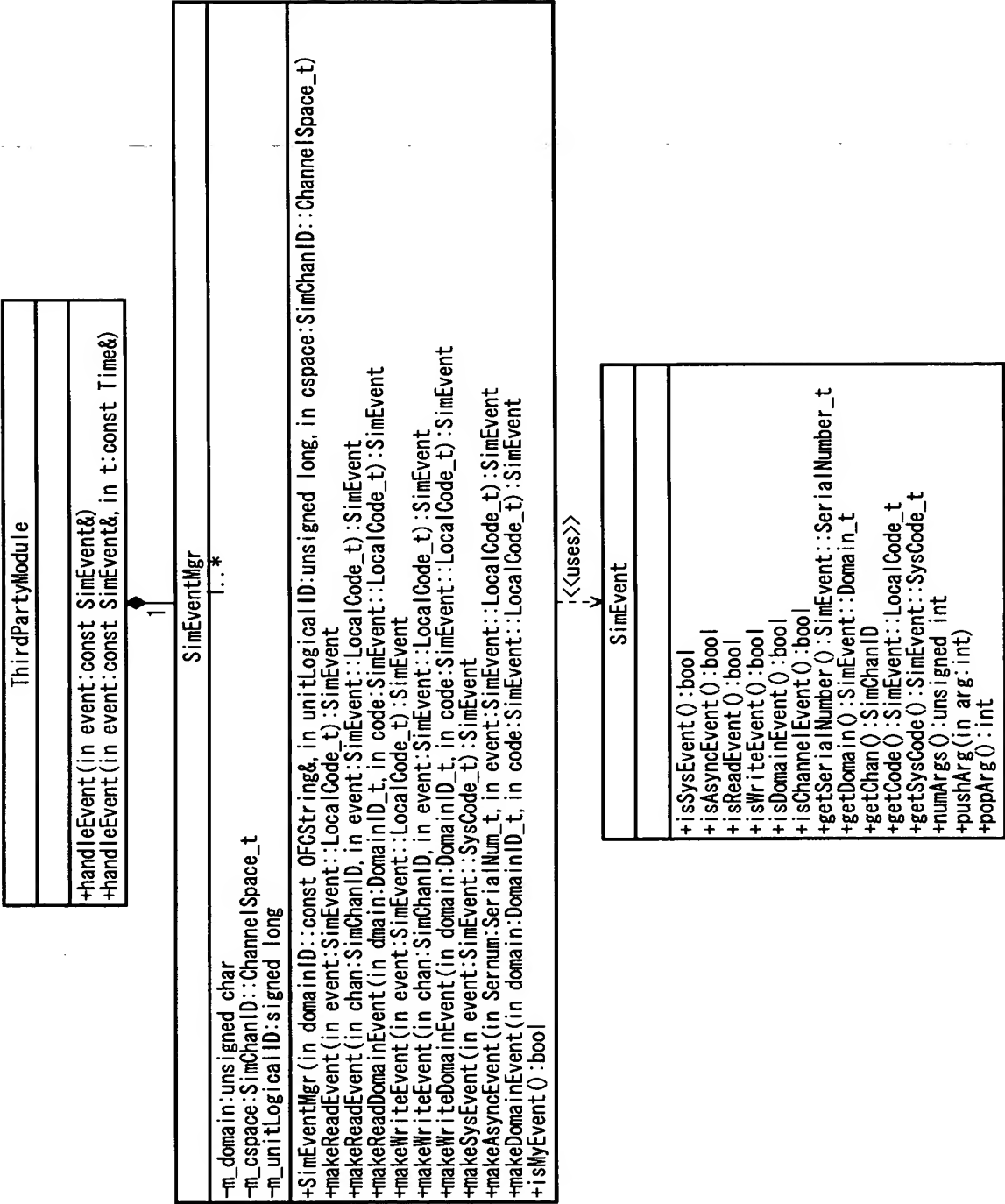


FIG. 20

```

#include "OAI/sim/SimComponent.h"
#include "OAI/sim/SimChannel.h"
#include "OAI/sim/SimChanID.h"

#include "OAI/OFC/OFCString.h"

class MyBaseModule : public OASIS::SimComponent
{
public:
    // constructor
    MyBaseModule(unsigned int sernum, unsigned int prodrev,
                  unsigned int port);

    // channel access
    OASIS::SimChannel* getChannel(const OASIS::SimChanID& chan_id);

    // do nothing for these initialization steps
    void initialize() {}
    void initLoadEvents() {}

    // ID
    unsigned int getSerialNumber() { return m_sernum; }
    unsigned int getProductRevision() { return m_prodrev; }

    // bus I/O
    void setBaseAddress(unsigned int index, unsigned int base);
    void getBaseAddress(unsigned int index, unsigned int* base);
    void getModuleIDs(unsigned int* vendorID, unsigned int* moduleID);
    void setBusNumber(unsigned int number);
    void clearInterrupt();
    void lockInterrupt();
    void unlockInterrupt();
    ClassCode_t getClassCode();
    void read(unsigned int addr, unsigned int* data);
    void write(unsigned int addr, unsigned int data);

    // pattern tracing features (unimplemented)
    void poll(unsigned int&, Time& t, unsigned int) {}
    void poll(unsigned __int64&, Time& t, unsigned int) {}
    void poll(unsigned double&, Time& t, unsigned int) {}
    void poll(unsigned OFCString&, Time& t, unsigned int) {}

protected:
    // internal types
    typedef unsigned int port_t;
    typedef unsigned int reg_t;
    typedef unsigned char vector_t;
    typedef unsigned char vecaddr_t

    // internal data
    OASIS::SimChannel m_channels[8];
    reg_t m_sernum;
    reg_t m_prodrev;
    port_t m_port;
    reg_t m_base;
    reg_t m_period;
    reg_t m_edges[8];
    reg_t m_high;
    reg_t m_low;
    vector_t m_pattern[100];
    vecaddr_t m_patPtr;
    OASIS::SimEventMgr m_evtMgr;
    bool m_intrLock;
};

```

```
#include "MyBaseModule.h" // base class header

class MyDriveModule : public MyBaseModule
{
public:
    // constructor
    MyDriveModule(unsigned int sernum, unsigned int prodrev,
                  unsigned int port);

    // access to vendor/module data
    void getModuleIDs(unsigned int* vendorID, unsigned int* moduleID);

    // initialize simulation
    void initEvents();

    // handle events
    void handleEvent(const OASIS::SimEvent& event);
    void handleEvent(const OASIS::SimEvent& event, const OASIS::Time& t);
};
```

*FIG. 22*

```

void MyDriveModule::handleEvent(const OASIS::SimEvent& event,
                                const OASIS::Time& t)
{
    // t is the start of the cycle since that is the time we
    // registered for

    // write the edge into each channel
    for (chanaddr_t c = 0; c < 8; c++)
    {
        // write the edge to the output channel
        m_channels[c].set(t + m_edges[c] * 1.0e-12,
                          (m_pattern[m_patPtr] & 1 << c ? m_high : m_low)
                          * 1.0e-3);

        // terminate the write
        m_channels[c].end(t + m_period * 1.0e-12);
    }

    // register for the next start of cycle
    if (m_patPtr++ < 100)
    {
        registerEvent(m_evtMgr.makeWriteEvent(DRIVE_CYCLE),
                      t + m_period * 1.0e-12);
    }
}

```

*FIG. 23*

```

#include "MyBaseModule.h" // base class header

class MyStrobeModule : public MyBaseModule
{
public:
    // constructor
    MyStrobeModule(unsigned int sernum, unsigned int prodrev,
                    unsigned int port);

    // access to vendor/module data
    void getModuleIDs(unsigned int* vendorID, unsigned int* moduleID);

    // bus I/O
    void read(unsigned int addr, unsigned int* data);
    void write(unsigned int addr, unsigned int data);

    // initialize simulation
    void initEvents();

    // handle events
    void handleEvent(const OASIS::SimEvent& event);
    void handleEvent(const OASIS::SimEvent& event, const OASIS::Time& t);

private:
    // internal types
    typedef unsigned char chanaddr_t;

    // fail vector memory
    chanaddr_t m_fcm[10];
    vecaddr_t  m_fvm[10];
    reg_t      m_failCnt;
};

```

*FIG. 24*



25/28

```

void MyStrobeModule::handleEvent(const OASIS::SimEvent& event,
                                const OASIS::Time& t)
{
    // t is the end of the cycle since that is the time we
    // registered for

    // compute the start of cycle since all edges are relative to it
    OASIS::Time t0 = t - m_period * 1.0e-12;

    // strobe each channel
    for (chanaddr_t c = 0; c < 8; c++)
    {
        // make sure fail vector memory is not full
        if (m_failCnt == 10)
        {
            return;
        }

        // strobe the input channel
        Voltage v = m_channels[c].read(t0 + m_edges[c] * 1.0e-12);

        // test high
        if (m_pattern[m_patPtr] & 1 << c)
        {
            // fail
            if (v < m_high * 1.0e-3)
            {
                m_fcm[m_failCnt] = c;
                m_fvm[m_failCnt++] = m_patPtr;
            }
        }

        // test low
        else
        {
            // fail
            if (v > m_low * 1.0e-3)
            {
                m_fcm[m_failCnt] = c;
                m_fvm[m_failCnt++] = m_patPtr;
            }
        }

        // terminate the read
        m_channels[i].flush(t);
    }

    // register for the next end of cycle
    if (m_patPtr++ < 100)
    {
        registerEvent(m_evtMgr.makeReadEvent(STROBE_CYCLE),
                     t + m_period * 1.0e-12);
    }
}

```

*FIG. 25*

```

#include "SimComponentStepped.h"
#include "SimChannel.h"
#include "SimChanID.h"
#include "OASISEngTypes.h" // for Time and Voltage

class MyDUTModel : public OASIS::SimComponentStepped
{
public:
    // constructor
    MyDUTModel(const OASIS::Voltage& vih, const OASIS::Voltage& vil,
               const OASIS::Voltage& voh, const OASIS::Voltage& vol);

    // get channels
    SimChannel* getChannel(const OASIS::SimChanID& chan_id);

    // run method
    void run(const OASIS::Time& t0, const OASIS::Time& tf);

private:
    // channels
    OASIS::SimChannel m_inputs[8];
    OASIS::SimChannel m_outputs[8];

    // levels
    OASIS::Voltage m_vih;
    OASIS::Voltage m_vil;
    OASIS::Voltage m_voh;
    OASIS::Voltage m_vol;
    OASIS::Voltage m_vz;
};

```

*FIG. 26*

```

void MyDUTModel::run(const OASIS::Time& t0, const OASIS::Time& tf)
{
    // loop through each wire
    for (unsigned i = 0; i < 8; i++)
    {
        // loop through all the edges from the input channel
        for (OASIS::SimWaveformIter wf_iter
             = m_inputs[i].getWaveformIter(m_vih, m_vil, t0, tf);
             !wf_iter.end(); wf_iter.next())
        {
            // write edges with 1 nsec delay
            switch (wf_iter.getState())
            {
                case OASIS::SimWaveformIter::H:
                    m_outputs[i].set(wf_iter.getTime() + 1.0e-9, m_voh);
                case OASIS::SimWaveformIter::L:
                    m_outputs[i].set(wf_iter.getTime() + 1.0e-9, m_vol);
                case OASIS::SimWaveformIter::Z:
                    m_outputs[i].set(wf_iter.getTime() + 1.0e-9, m_vz);
            }
        }

        // terminate write for each output channel
        m_outputs[i].end(tf);

        // flush each input channel
        m_inputs[i].flush(tf);
    }
}

```

*FIG. 27*

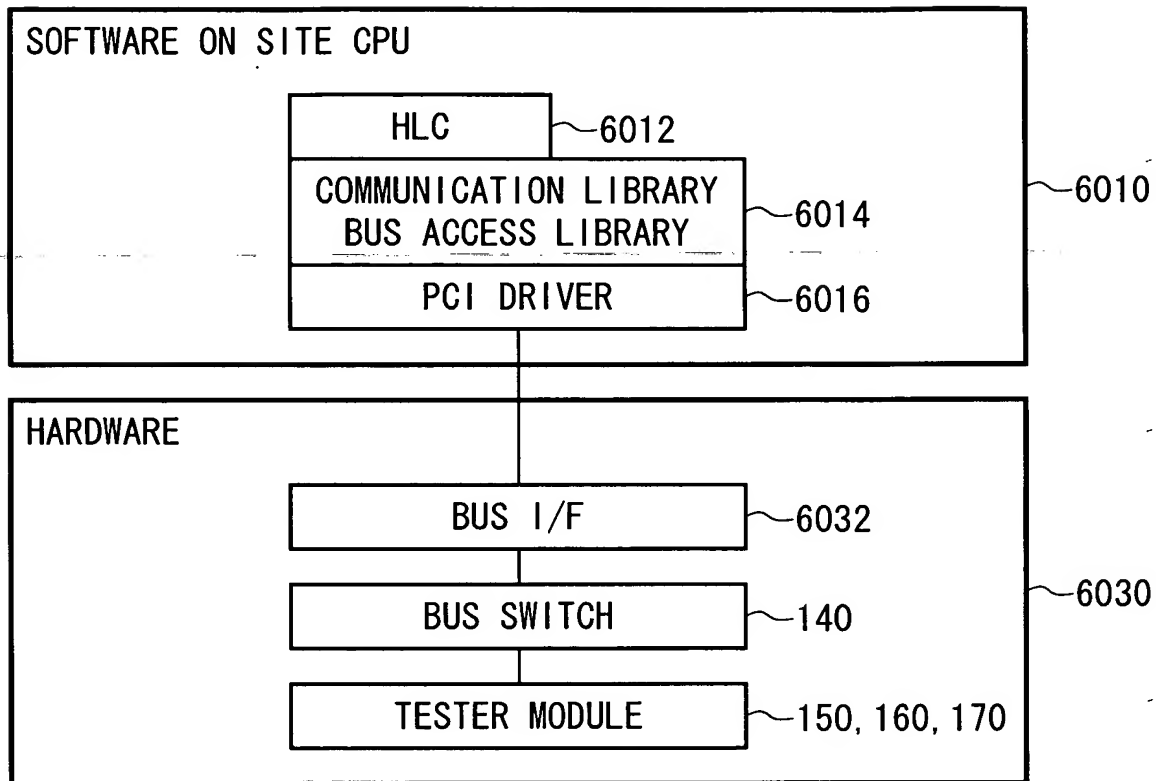
6000

FIG. 28A

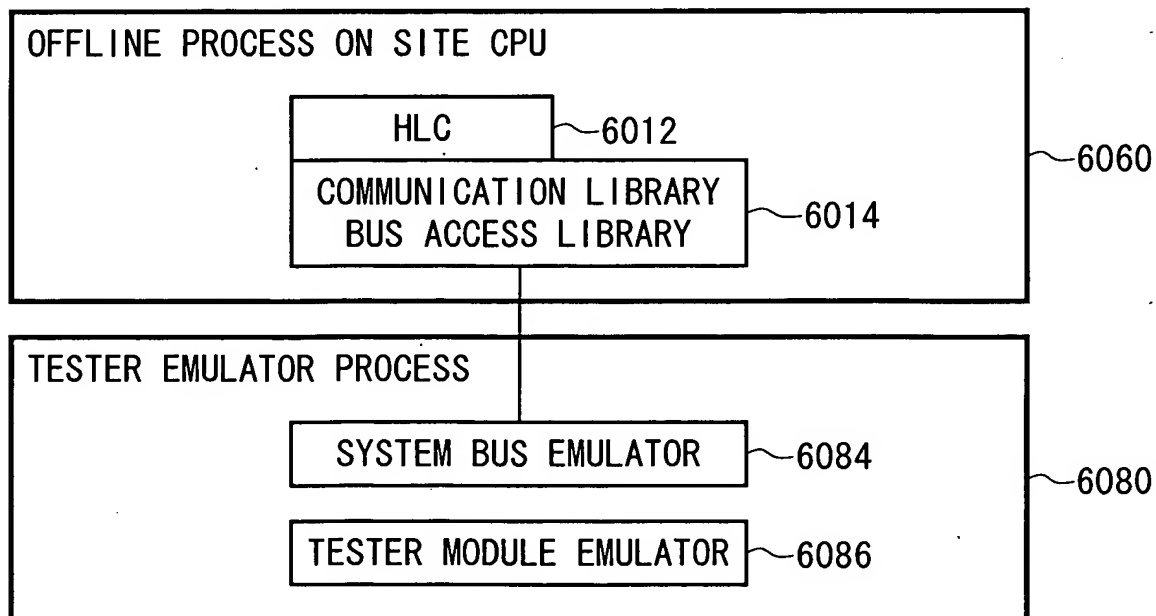
6050

FIG. 28B